Vili Ahonen

TEKOÄLYALGORITMIT VIDEOPELEISSÄ

TEKOÄLYALGORITMIT VIDEOPELEISSÄ

Vili Ahonen
Opinnäytetyö
Kevät 2018
Tietotekniikka tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä(t): Vili Ahonen
Opinnäytetyön nimi: Tekoälyalgoritmit videopeleissä
Työn ohjaaja(t): Eino Niemi
Työn valmistumislukukausi ja -vuosi: Kevät 2018
Sivumäärä: 39

Tässä opinnäytetyössä tutkittiin, minkälaisia erilaisia videopelien tekoälytyyppejä tunnetaan, millaisissa tilanteissa niitä voidaan soveltaa ja minkälaisia algoritmeja voidaan käyttää videopelitekoälyjen toteuttamisessa. Lisäksi toteutettiin tekninen toteutus, joka käsittelee päätöksentekoa sekä polunetsintää.

Opinnäytetyö suoritettiin omien intressien pohjalta ja omasta toimeksiannosta. Tekninen toteutus toteutettiin Unity3D-ympäristössä. Toteutus sisältää kaksi osaa: Ant, joka käsittelee päätöksentekoa ja Adventurer, joka käsittelee polunetsintää.

Asiasanat: Tekoäly, videopelit, peliohjelmointi, polunetsintä-algoritmit

ABSTRACT

Oulu University of Applied Sciences
Information Technology, degree of program

Author(s): Vili Juhani Ahonen
Title of thesis: Artificial Intelligence algorithms in video games
Supervisor(s): Eino Niemi
Term and year when the thesis was submitted: Spring 2018
Pages: 39

This thesis aims to research difference between artificial intelligences in video games and study applications for them. Work includes a technical product which focuses on choice making and pathfinding.

Thesis is based on my own interest and no company is involved in development. The technical product is in two parts. Ant involves choice making algorithm and Adventurer involves pathfinding algorithm. This technical product is supposed to work as a good base for my future AI projects and as well a good introduction to those who are interested in AI development.

Keywords: Artificial intelligence, video games, game programming, pathfinding algorithms

SISÄLLYS

SANASTO

A*	A star, polunetsintä algoritmi
AI	Tekoäly, englannin kielen lyhenne *Artificial Intelligence*
Algoritmi	Ohje tai kaava suoritettavasta prosessista
Boolean	Booleanin logiikan mukainen muuttuja, joka on tosi tai epätosi (Beal 2018)
Idle	Tila jossa ei tehdä mitään tai odotellaan
Immersio	Viitataan videopeliaiheissa pelaajan syvään keskittymiseen ja ulkopuolisen maailman hetkelliseen unohtamiseen (Immersio. 2017)
Narratiivi	Ennalta määrätty kertomus (Narratiivi. 2018)
Node	Paikannuspiste peliympäristössä.
Nopeus	Tietyn ajan sisällä edetyn matkan pituus ja suunta.
Skripti	Komentokoodi pelihahmoille ja -objekteille
Pathfinding	Polunetsintäalgoritmi, joka etsii mahdollisimman hyvä reitin paikasta A paikkaan B ja ottaa huomioon tarvittavat muuttujat (Polunetsintä. 2013)
Parametri	Välitettävä tieto esimerkiksi algoritmille.
UI	User Interface, kaikki vuorovaikutettavat näppäimet ja paneelit mitä käyttäjä näkee näytöllä

1 JOHDANTO

Käsittelen tässä opinnäytetyössä tekoälyjen esiintymistä videopeleissä ja niiden eri menetelmiä. Työssä käsittelen abstrakteja aiheita, kuten esimerkiksi miten tekoälyt luovat realistisen ympäristön, jolloin teoreettista lähdetietoa on niukasti tai ei ole saatavilla lainkaan. Osa tekstin esimerkkitapauksista pohjautuu omiin kokemuksiini videopelien parissa ja näin ollen kuvaavat omia tulkintojani.

Olen ollut kiinnostunut tekoälyistä jo kauan ja olen suuri tieteisfiktion ja kyberpunkin ihailija, millä oli vaikutusta aiheen valinnassa. Kokemukseni tekoälyjen tutkimisesta on ollut pääsääntöisesti videopelejä pelatessa NPC-hahmojen – eli ei-pelaajahahmojen (engl. non-player-character) – käytöksen seuraamista pelaajan liikehdinnän ja toimintojen mukaan. Työ ei käsittele oppivia koneita tai neuroverkkojen käyttöä peliohjelmoinnissa. Työ sisältää myös kaksi erilaista teknistä toteutusta. Kehitysalustaksi valitsin Unity-pelimoottorin v. 2017.3.0f3 Personal, koska se on minulle tuttu työkalu. Ohjelmakoodin tuottamiseen käytän Microsoft Visual Studio Community 2015 versio 14.0.25431.01:tä.

Kokemukseni peliohjelmoinnista koostuu kolmesta kouluprojektista, useammasta vapaa-ajan projektistani sekä yhdestä Oulu Game Lab-projektista. Roolini OGL-projektissa oli ohjelmoija, mutta toimin myös pelisuunnittelijana. Tässä työssä kaikki tekoälyihin liittyvä teoria sekä käytäntö käsitellään videopeleissä käytettävien tekoälyjen näkökulmasta.

2 TEKOÄLYT VIDEOPELEISSÄ

Tekoäly – tunnetaan myös lyhenteellä AI – määritellään tietotekniikan osa-alueeksi, joka käsittelee tietokoneiden älykästä toimintaa ihmisen tavoin. Tekoälyjen kehityksessä huomioon otettavia näkökulmia ovat tietoisuus, päättely, loogisten ongelmien ratkaisu, huomiointi, oppiminen, suunnittelu sekä objektien manipulointi. (Artificial Intelligence (AI). 2018.)

2.1 Tekoälyjen soveltaminen videopeleissä

Tekoälyä voidaan verrata laajaan algoritmisettiin, joka lainaa käytäntöjä säätötekniikasta, robotiikasta, tietotekniikasta ja tietokonegrafiikasta, jolloin videopeleissä käytettäviä tekoälyjä ei luonnehdita todellisiksi tekoälyiksi. Tälle argumenttina toimii videopelitekoälyjen toiminnan keskittyminen rajallisiin ja ennalta asetettuihin vaihtoehtoihin, joiden kanssa tekoäly toimii rajallisten vuorovaikutuksien kautta. Tekoälyjä käytetään videopeleissä luomaan kuvitelman älykkäästä käyttäytymisestä, ja ne poikkeavat näin perinteisestä tekoälyjen määritelmästä. (Artificial Intelligence (video games). 2018.)

Omien tulkintojeni ja kokemuksieni pohjalta olen huomannut, että videopeleissä tekoälyjen käyttö on vakiintunut genreille, joissa pelaaja pelaa tietokonetta vastaan ilman ihmisvastusta. Pelin tekoäly huolehtii pelaajan vastustamisesta ja hankaloittaa – tai jopa estää – tavoitteiden saavuttamista. Tekoäly voi esimerkiksi ohjata vihollishahmoa kohti pelaajahahmoa ja suorittamaan hyökkäyksen, ohjata vihollishahmoa poispäin pelaajahahmosta paetakseen tai ennakoida pelaajahahmon seuraavan mahdollisen liikkeen ja mukauttaa omat tulevat päätöksensä tähän ennalta arvioituun dataan. Samaan aikaan useammalle pelaajalle tarkoitetuissa videopeleissä ei yleensä ole tarvetta vastustaja-tekoälyille, koska pelaajat toimivat vastustajina toisilleen, mutta peli voi sisältää pienempiä mikrotekoälyjä. Esimerkiksi strategiapeleissä NPC-hahmot käyttävät pathfinding-algoritmia löytääkseen lyhimmän reitin pelaajan haluamaan kohteeseen, mutta ne osaavat käydä hyökkäykseen, jos kohtaavat matkalla vihollisen.

Tulkintojeni ja kokemuksieni pohjalta olen huomannut, että tekoälyillä voidaan myös luoda realismia pelimaailmaan. Kaupungit tarvitsevat jalankulkijoita sekä autoilijoita liik-

kumaan teillä ja luonto tarvitsee eläimiä vaeltelemaan ja käyttäytymään roolinsa mukaisesti. Kuvassa 1 on esimerkki ympäristörealismin rehevöittämisestä Grand Theft Auto -toimintapelisarjasta (Rockstar Games, 1997). Pelisarjan kaupungit tuntuvat realistisilta ympäristön tekoälyn ansiosta. Jalankulkijahahmojen tarkoitus ei ole estää pelaajaa millään tavoin, mutta ne antavat pelaajalle rahaa ja resursseja kuollessaan. Jalankulkijoiden eliminointi kuitenkin nostattaa pelaajan rikollisuus-tasoa, johon peli reagoi synnyttämällä poliiseja, jotka pyrkivät ajamaan pelaajaa takaa sekä eliminoimaan hänet. Pelaajalle siis tarjotaan kerättäviä resursseja mutta syy-seurausperiaatteella, jonka pohjalta taustalla käyvä tekoäly toimii.

KUVA 1. Grand Theft Auto (Grand Theft Auto. 2018)

2.2 Erilaisten tekoälyjen toteutus videopeleissä

Tekoälyjen kompleksisuus vaihtelee peligenrekohtaisesti, sillä sen tarve määrittelee vahvasti sen laajuuden. Seikkailuhenkisissä peleissä tarvitaan useampia eri tavoin käyttäytyviä älyjä, jotta pelimaailma ja sen sisältö eivät rikkoisi immersiota eli taustalla vaeltelevat

NPC-kyläläiset noudattavat tiettyä skriptiä eli ohjetta, johon voi liittyä esimerkiksi liikkumista paikasta A paikkaan B. Viholliset voivat käyttää samankaltaista skriptiä, mutta pelaajan nähdessään ne vaihtavat tilansa taistelu-tilaan ja hyökkäävät pelaajan kimppuun. Tekoälyille on mahdollista luoda realistinen sykli vuorokausien ympärille, jos pelissä on vuorokauden vaihdosominaisuus. Kun päivä kääntyy pelissä kohti iltaa, kyläläiset suuntaavat ennalta määrätyille sijainneille ja siirtyvät idle-tilaan, joka muistuttaa pelaajan näkökulmasta lepäämistä tai odottelua. Seikkailupelien tekoälyt voivat olla todella vaihtelevia kompleksisuudeltaan. Edellä mainitut tapaukset muistuttavat todella paljon tilakoneen toimintaa (Digitaalipiirit/Tilakoneet. 2008). Voidaan todeta, että tapauksien hahmot käyttävät tilakonetta hyväkseen määrittämään, millaisia toimintoja hahmot voivat suorittaa

Ajopeleissä tietokoneen ohjaamien autojen ensisijainen tavoite on ajaa rataa pitkin parasta mahdollista reittiä ja saavuttaa maaliviiva ennen pelaajaa. Pelimaailmana ajopeleissä toimii usein lineaarinen ajorata, jossa on kaarteita ja esteitä. Kaarteissa autot yrittävät ajaa mahdollisimman lyhyttä reittiä, eli usein ajoradan sisäreunaa pitkin. Hidastavia tai pysäyttäviä objekteja autot pyrkivät välttämään tai ohittamaan. Jos pelaaja yrittää estää kilpailevan auton ohituksen, auton täytyy ohittaa pelaaja käyttäen uutta reittiä ilman että ajautuu liian kauas oikeasta suunnasta – tässä tapauksessa ajoradan ajosuunnasta. Autot myös pyrkivät tavoittamaan aina maaliviivan lyhyimpien ajoreittien tuloksena. Tämä muistuttaa Steering Behavior -nimistä algoritmien sarjaa, joka sisältää erilaisia käyttäytymisiä, esimerkiksi seuraamista ja väistelyä (Reynolds 1999).

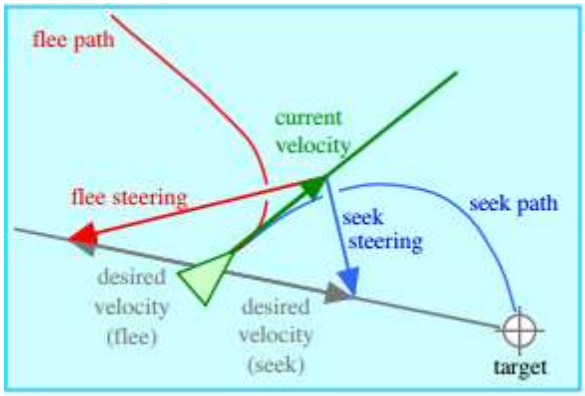
Oman tulkintani mukaan tekoälyä voidaan soveltaa myös päätösten tekoon videopeleissä. Vuoropohjaisissa strategiapeleissä tekoälyn täytyy pystyä reagoimaan pelaajan tekemiin toimintoihin ja toimia itse sen mukaan. Myös ennakointi on tärkeä osa strategista pelaamista, joten tekoälyn tulee pystyä valitsemaan tilanteisiin oikeat vaihtoehdot pärjätäkseen. Tappelupeleissä pelaajien tavoite on päihittää vastustajan lyönneillä, potkuilla, torjunnoilla ja erikoisliikkeillä. Vastustajahahmo seuraa pelaajahahmon liikkeitä ja käyttää näitä samoja tai samankaltaisia liikkeitä päihittääkseen pelaajan hahmon. Tämä muistuttaa myös tilakonepohjaista tekoälyä, mutta tilakoneen muutokset määräytyvät pelaajan liikkeiden mukaan. Pelaajan laskiessa suojauksen tilakone muuttuu tilaan joka antaa vastustajahahmon hyökätä tilanteeseen parhaiten soveltuvalla hyökkäyksellä. Koska nopei-

den prosessorien ansiosta tietokoneet kykenevät suorittamaan useita toimintoja sekunneissa, tappelupeli tekoälyjen toimintoja on viivytettävä, sillä muuten vastustaja hahmon liikkeen tuntuisivat liian nopeilta, mikä taas tekisi pelistä epäreilun.

2.3 Steering Behavior -algoritmit

Steering behaviorilla tarkoitetaan liikealgoritmien sarjaa, jossa algoritmit käsittelevät kappaleen vauhtia ja suunnan muutosta. Useimmilla näillä algoritmeilla on samankaltainen rakenne: niiden käytös määräytyy jonkin kohteen liikkeen tai rajoitetun datan mukaan. Esimerkiksi takaa-ajoon ja väistelyyn liittyy yleensä jokin toinen objekti ja esteiden välttelyssä käytetään peliympäristön geometriaa. Jokainen algoritmi suorittaa yhden käyttäytymistä vastaavan funktion, joten algoritmeja voidaan pinota päällekkäin. Tällöin algoritmit työskentelevät sulavasti yhdessä, eivätkä häiritse toisiaan. (Millington – Funge 2009, 55-56.)

Tarkastellaan Craig W. Reynoldsin toteutus menetelmiä eri liikealgoritmeista dokumentissa Steering Behaviors for Autonomous Characters (Reynolds 1999). Yksinkertaisia algoritmeja ovat mm. **Seek**- ja **Flee**-algoritmit. Seek-algoritmi ohjaa liikkuvaa objektia – kuvitellaan että esimerkissämme on auto – tiettyä paikannuspistettä tai kohdetta kohti säädellen samalla sen nopeutta etäisyyden suhteen. Flee-algoritmi toimii päinvastaisella periaatteella, eli liikkuva objekti liikkuu päinvastaiseen suuntaan. Seek-algoritmia ei tule kuitenkaan sekoittaa tai rinnastaa painovoimaa simuloivaan voimaan, koska algoritmin tarkoitus ei ole synnyttää kiertoratamaista liikettä kohteen ympärille. Kuvassa 2 kuvataan vihreän objektin eri nopeuksia. On otettava huomioon, että käytetään termiä nopeus suunnan sijaan, sillä algoritmi käsittelee kappaleen vauhtia ja suuntaa, jolloin lopputuloksena on nopeus. (Reynolds 1999.)



KUVA 2. Seek- ja Flee-algoritmien toteutus (Reynolds 1999)

Kuvassa vihreä kolmio kuvastaa liikkuvaa objektia ja objektin kohteena toimii ”target”. Objektin Seek-algoritmin haluama nopeus kuvataan ”desired velocity”-vektorina harmaalla värillä. Haluttu nopeus on määrätty objektin suurimman sallitun nopeuden mukaisesti. Kuvan ”seek steering” on erotus objektin nopeudesta ja halutusta nopeudesta. Kun algoritmin tekemä nopeuden muutos otetaan käyttöön, kappale liikkuu sinistä ”seek path”-linjaa pitkin kohteeseen. Kuten aikaisemmin mainittiin, Flee-algoritmi toimii päinvastaisella tavalla halutun nopeuden suhteen ja tällöin kappale liikkuu kuvan mukaista punaista linjaa pitkin poispäin kohteesta. Seek-algoritmia ei ole tarkoitettu käytettäväksi tilanteissa, joissa kappale saavuttaa kohteen tarkan sijainnin, sillä saavuttamisen hetkellä kappale jatkaa matkaa, kääntyy takaisin ja yrittää saavuttaa kohteen uudelleen. (Reynolds 1999.) Reynolds antaa dokumentissaan kaavan tällaiselle käytökselle:

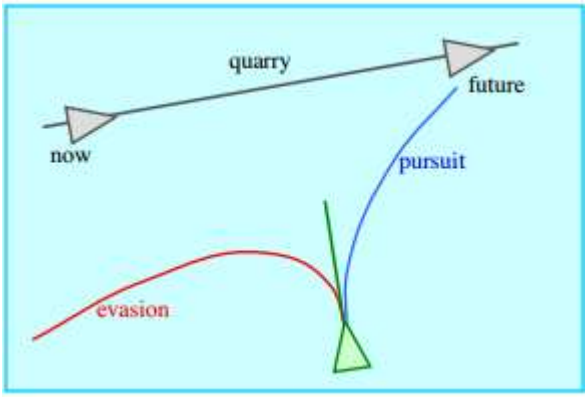
```
desired_velocity = normalize (position - target) * max_speed

steering = desired_velocity – velocity
```

Haluttu nopeus saadaan, kun normalisoidaan hakeutuvan objektin ja kohteen sijoituksen erotus ja kerrotaan se maksiminopeudella. Lopullinen ohjautuminen saadaan, kun halutusta nopeudesta vähennetään hakeutuvan objektin oma nopeus. (Reynolds 1999.)

Pursuit- ja **Evasion**-algoritmit ovat toiminnaltaan samankaltaisia kuin Seek ja Flee. Pursuit-algoritmin merkittävin eroavaisuus suhteessa Seek-algoritmiin on liikkuva kohde ja sen sijainnin arviointi suhteessa aikaan. Kohteen tuleva sijainti arvioidaan ja tähän arvioituun sijaintiin hakeudutaan Seek-algoritmin tavoin nopeuden muutoksilla. Evasion-al-

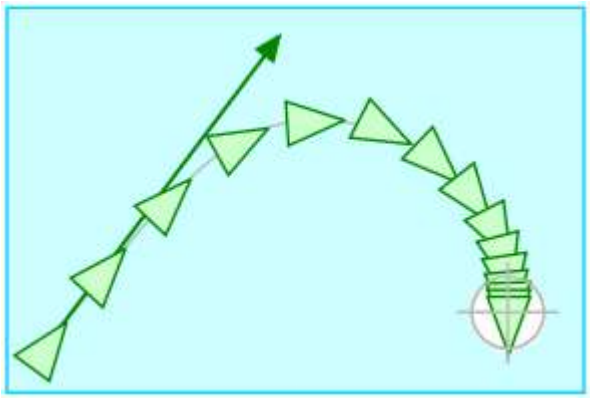
goritmi toimii vastaavalla tavalla kuin Pursuit, mutta poikkeuksellisesti käyttää Flee-algoritmia hyväkseen hakeutumaan poispäin liikkuvan kohteen arvioidusta sijainnista. (Reynolds 1999.)

Kuvassa 3 harmaana kolmiona esitetty kohde liikkuu ennalta määrättyä reittiä pitkin ja Pursuit-algoritmi suorittaa hakeutumisen sen nopeuden suuntaisesti ja Evasion-algoritmi suorittaa liikkeen päinvastoin. On huomioitava, että Evasion-algoritmi on vaihtoehtoisempi suhteessa kohteen nopeuteen. Kuvan ”evasion”-reitti voi poiketa, jos harmaan kohteen nopeus olisi lähestyvä tai erkaneva vihreän objektin nopeuden suunnan suhteen. (Reynolds 1999.)



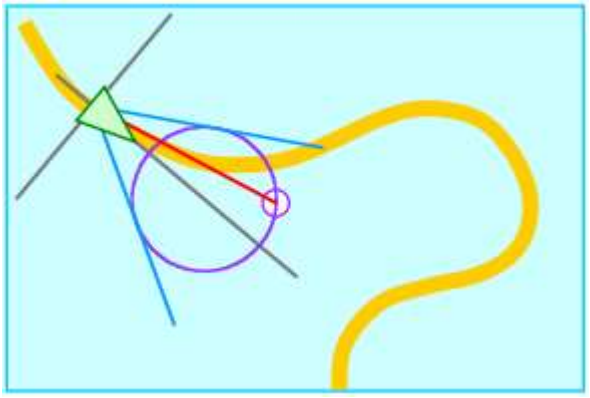
KUVA 3. Pursuit- ja Evasion-algoritmien toteutus (Reynolds 1999)

Arrival-algoritmi toimii Seekin tavoin, mutta poikkeaa siitä kiihtyvyyden muutoksella sekä etäisyysparametrilla, joka on Arrival-algoritmille oleellinen arvo. Parametri määrittää, kuinka lähelle kohdetta objekti haluaa päästä, ikään kuin tyytyväisyysetäisyys. Arrival-algoritmi poikkeaa Seek-algoritmista silloin, kun objekti lähestyy kohdetta. Arrival-algoritmi laskee objektin kiihtyvyyttä suhteessa etäisyyteen ja objekti saapuu hitaasti tyytyväisyysetäisyydelle. Kuvassa 4 vihreänä kuvattu objekti hidastaa vauhtiaan saapuessaan lähemmäs parametrin määrittämään etäisyyttä. Saavutettuaan etäisyyden objekti pysähtyy. Ohjautumiseen tässä käytetään hyväksi Seek-algoritmia. Arrival-algoritmi toimii yleisesti saavuttamiseen, toisin kuin Seek ja Pursuit, jotka liikkuvat kohteen ohi ja yrittävät aina vain uudestaan saavuttaa kohteen. (Reynolds 1999.)



KUVA 4. Arrival-algoritmin toteutus (Reynolds 1999)

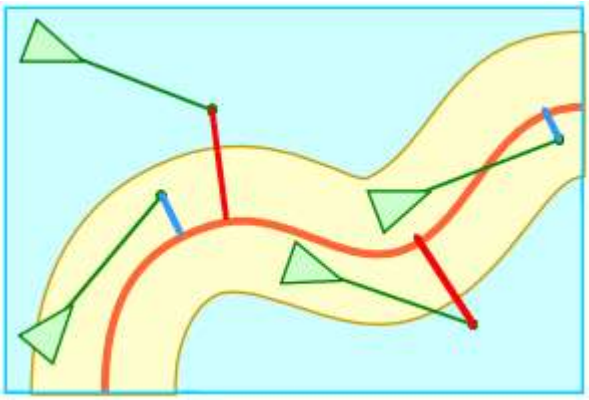
Wanderer-algoritmia käytetään satunnaisen liikkeen toteuttamiseen. Algoritmi voidaan luoda usealla eri tavalla, esimerkiksi antamalla objektille joka sekunti uusi satunnainen suunta. Tämä ei tietenkään tuottaisi realistista vaeltelua, vaan nykivää edestakaisin seilaamista. Algoritmi voidaan myös luoda siten, että objektille annetaan satunnaisesta suunnasta piste, johon objekti sitten hakeutuu kuvan 5 mukaisesti. (Reynolds 1999.) Tässä voidaan soveltaa myös Arrive-algoritmia siten, että objekti hakeutuu vaihtelevalla kiihtyvyydellä määränpäähänsä. Tämän jälkeen arvotaan uusi suunta ja kohde. (Reynolds 1999.)



KUVA 5. Wanderer-algoritmin yksi toteutusmenetelmä (Reynolds 1999)

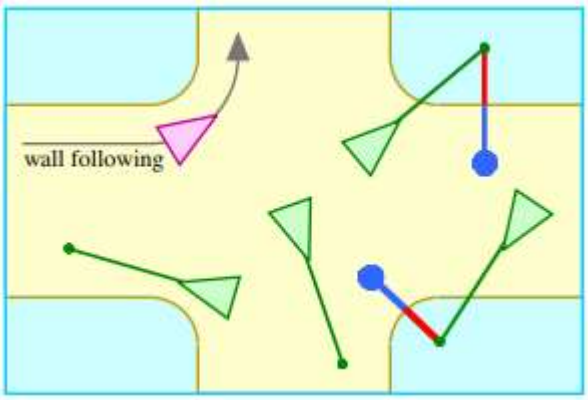
Path following -algoritmia käytetään ohjaamaan objektia ennalta määrättyä reittiä, kuten tietä, raiteita tai tunnelia pitkin. Algoritmin käyttöönotossa voidaan kuvitella, että objekti kulkee lankaa pitkin, jonka ympärillä on sylinterin muotoinen tila. Tämä tila määräytyy säteellä, joka mitataan langasta ja kattaa koko langan pituuden. Algoritmin avulla objekti pyrkii liikkumaan sylinterin sisällä ilman, että koskaan poistuu sen sisältä. Pysyäkseen

sisällä objektin täytyy hakeutua takaisin sylinterin sisälle. Tila toimii ikään kuin ohjaajana ja tienviitoittajana objektille. Kuvassa 6 kuvataan siniset ja punaiset viivat kuvaavat vihreän objektin nopeuden muutosta. (Reynolds 1999.)



KUVA 6. Path followingin toteutus (Reynolds 1999)

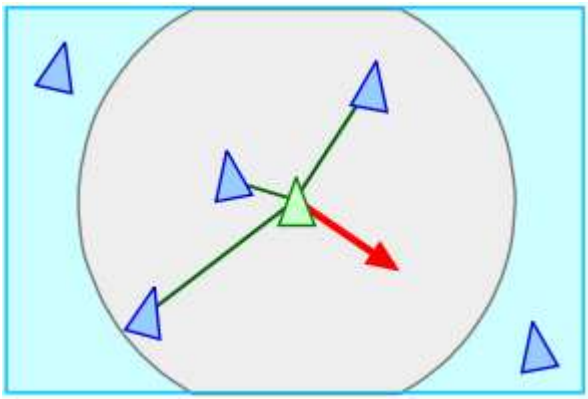
Path followingin variaatioita ovat **wall following** ja **containment**. Wall following tarkoittaa seinän tai muun pinnan lähestymistä ja etäisyyden ylläpitämistä kuvan 8 mukaisesti. **Containment** tarkoittaa käytännössä rajoitettua aluetta jonka sisällä objektien on pyrittävä liikkumaan. Kuvassa 7 vaaleanpunainen objekti liikkuu wall following -menetelmän mukaisesti ja vihreät objektit liikkuvat containment-menetelmän mukaisesti. Punaiset viivat ovat etäisyyksiä rajoitetun alueen reunoista ja siniset pisteet ovat tuleva sijainti johon halutaan hakeutua. (Reynolds 1999.)



KUVA 7. Wall following ja containment toteutus (Reynolds 1999)

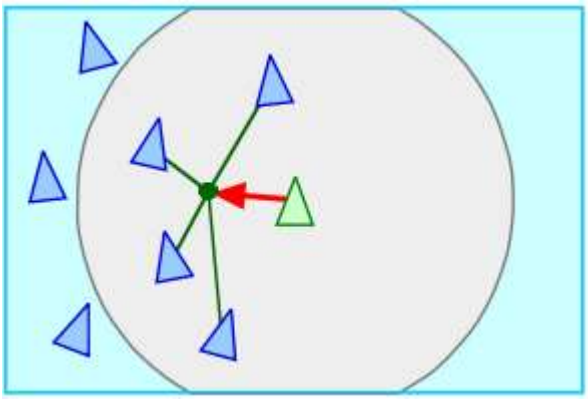
Separation, cohesion ja **alignment** ovat useamman objektin yhdistäviä algoritmeja. Jokainen algoritmi käsittelee yhden objektin reagointia sitä ympäröiviin naapuriobjekteihin.

Tämän objektipiirin ulkopuolisilla objekteilla ei ole vaikutusta näihin objekteihin. **Separation**-algoritmin perimmäinen tarkoitus on säilyttää objektin etäisyys muihin naapuriobjekteihin kuvan 8 mukaisesti sekä estää objektien pakkaantuminen. Toiminnallisesti objektille etsitään lähiympäristöstä muut vuorovaikutuskelpoiset objektit, jonka jälkeen objektiin kohdistetaan voima, joka ajaa sitä poispäin muista separation-algoritmin ohjaamista objekteista parametrin mukaiselle etäisyydelle. Punainen nuoli osoittaa vihreän objektin suunnan, kun voimat kohdistetaan vihreään objektiin. (Reynolds 1999.)



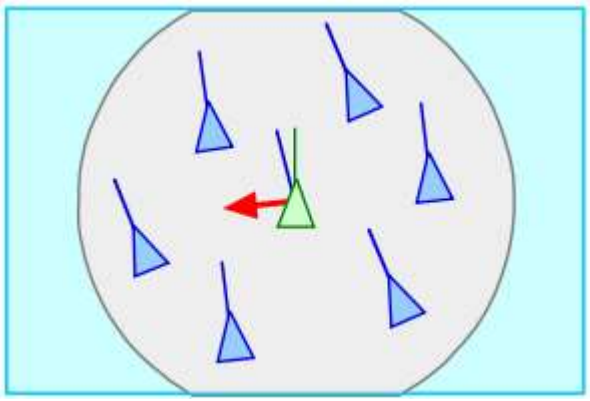
KUVA 8. Separation-algoritmin toteutus (Reynolds 1999)

Cohesion-algoritmi toimii vastapainona separate-algoritmille pitämällä objektit tarpeeksi lähellä toisiaan. Algoritmin toteutuksen alku suoritetaan separation-algoritmin tavoin etsimällä ensin lähipiirin naapuriobjektit, mutta voima kohdistetaan kohti kaikkien naapuriobjektien kattavan alueen keskipistettä. (Reynolds 1999.) Cohesion-algoritmia voidaan käyttää esimerkiksi ryhmittäytymisessä ja muodostelman muodostamisessa. Kuvassa 9 vihreällä kuvattu objekti pyrkii punaisen nuolen osoittamaan suuntaan. (Reynolds 1999.)



KUVA 9. Cohesion-algoritmin toteutus (Reynolds 1999)

Alignment-algoritmilla objekti saadaan käännettyä samansuuntaiseksi suhteessa sitä ympäröiviin objekteihin. Toiminnallisesti objekti etsii Separation ja Cohesion -algoritmien tapaan lähimmät naapuriobjektit ja yrittää täsmätä oman nopeutensa näiden kanssa. Kuvassa 10 vihreä objekti saa naapureiltaan keskiarvonopeuden, joka kuvataan sinisenä viivana. Tämä keskiarvo nopeus on objektin haluama suuntaus. Punainen nuoli kuvaa objektiin kohdistettavan voiman suunnan ja vihreällä viivalla kuvailtu objektin nykyistä suuntaa. (Reynolds 1999.)



KUVA 10. Alignment-algoritmin toteutus (Reynolds 1999)

Separation-, cohesion- ja alignment-algoritmeilla voidaan toteuttaa uniikki **Flocking**-kokoanisuus eli parveilu. Tämä on hyödyllinen käytäntö esimerkiksi pelimaailman eläinten käytöksen toteutuksessa. Parveilussa jokainen objekti pysyttelee asetettujen rajojen puitteissa, joita voivat olla esimerkiksi minimietäisyys, maksimietäisyys, samansuuntaisuus, erisuuntaisuus jne. (Reynolds 1999.)

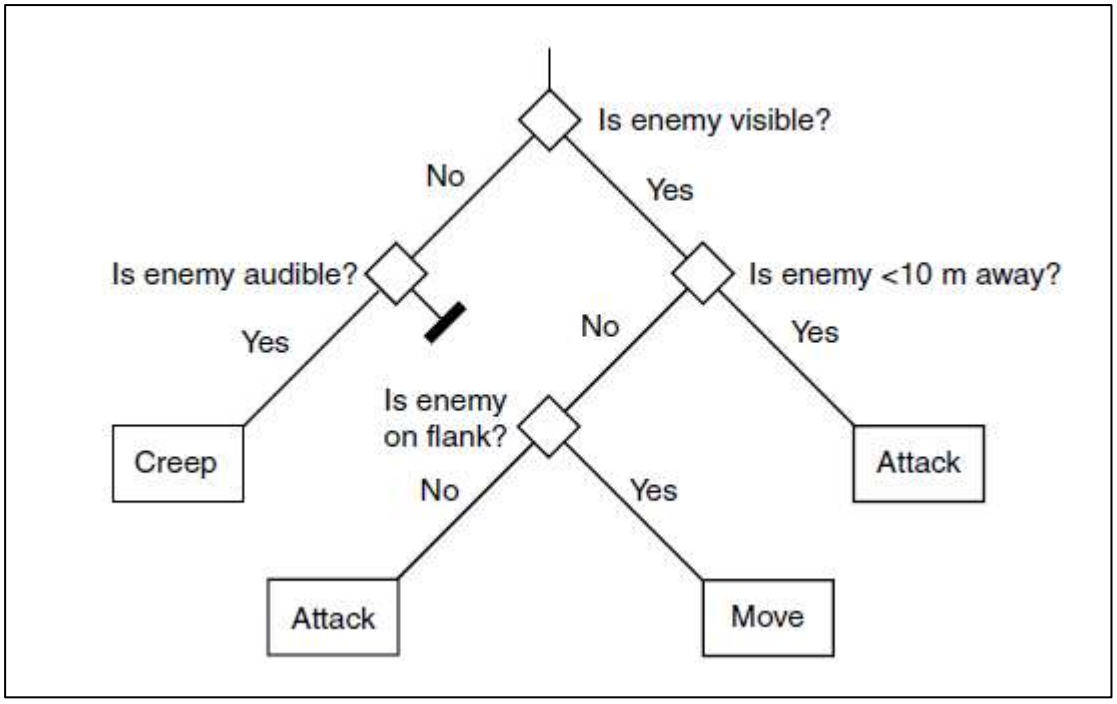
Edellä mainitut yksinkertaiset algoritmit toimivat rakennuspalikoiden tapaan yhteen sopivina osina, joilla voidaan toteuttaa suurempia kokonaisuuksia. Jotta voidaan toteuttaa todentuntuisia ja luonnollisia toimintaeleitä hahmoille, täytyy mahdollistaa näiden rakennuspalikoiden toiminnan saumattomuus. Näiden palikoiden yhdistäminen voidaan toteuttaa kahdella tavalla. Ensimmäinen menetelmä on, että hahmo vaihtaa käyttäytymistään – eli ensisijaista algoritmia – ympäristön tapahtumien mukaisesti. Esimerkiksi saaliseläinlauma, joka aistii lähestyvän metsästäjän, vaihtaa aikaisemmin asetetun algoritmin nyt pakenevaan algoritmiin. Algoritmeja ei suoranaisesti yhdistetä tässä tilanteessa, vaan asetetaan tärkeysjärjestykseen. Saaliseläimet eivät jää suorittamaan aikaisempia toimintoja samalla kun pakenevat. Kun lauma pääsee tarpeeksi kauas metsästäjästä, lauma rauhoittuu ja lakkaa pakenemasta. Toinen menetelmä mahdollistaa kahden algoritmin

päällekkäisyyden. Saaliseläinten tulee pakenemisen aikana väistellä vastaan tulevia esteitä, joten nämä kaksi algoritmia joutuvat toimimaan toistensa parissa. Kumpaakin menetelmää voidaan käyttää keskenään. (Reynolds 1999.)

2.4 Decision Making -algoritmit

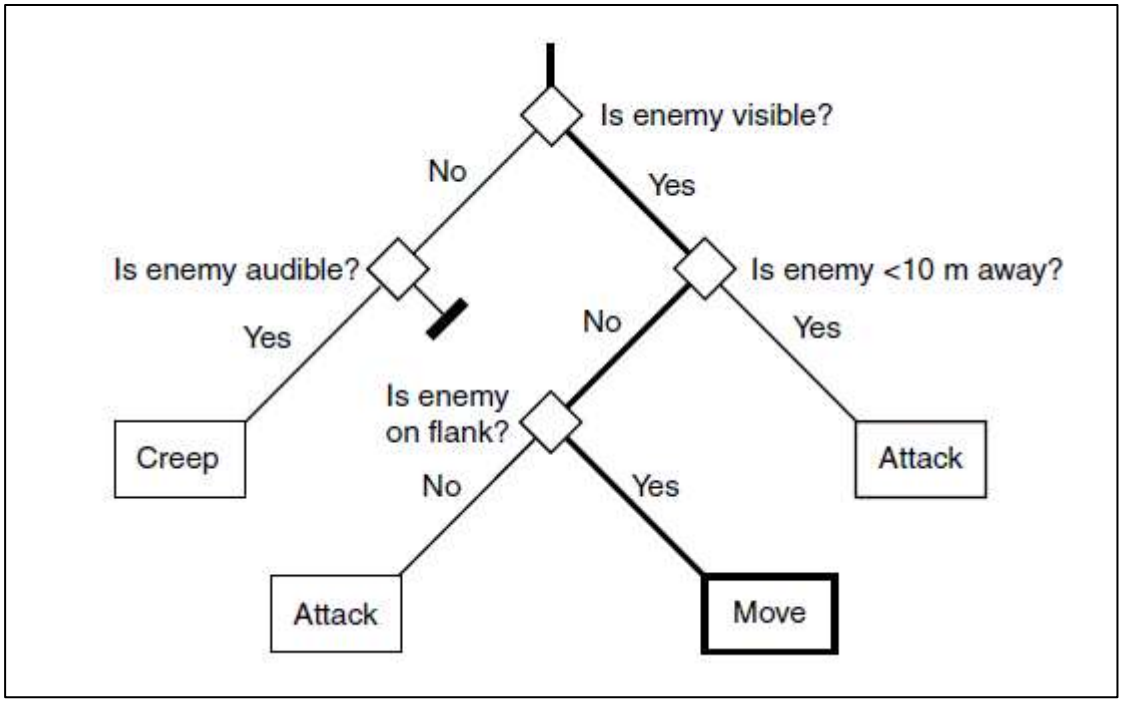
Decision making eli päätöksenteko on tyypillisesti osa tekoälyn kokonaisuutta. Useimmat pelit käyttävät yksinkertaisia ääreistilakoneita ja päätöspuita päätöksenteon toteuttamiseen. (Millington – Funge 2009, 293.) Tässä luvussa käsitellään **Decision Trees**-, **State Machines**-, **Fuzzy Logic**-, **Goal-oriented Behaviour**- ja **Rule-Based System**-menetelmiä.

Decision Trees eli päätöspuut ovat nopeita, helposti implementoitavia ja helppolukuisia ohjelmointirakenteita, joita käytetään usein hahmojen ja hahmojen animaatioiden ohjaamiseen. Päätöspuita voidaan soveltaa myös neuroverkko-oppimisessa, mutta sitä ei tässä työssä käsitellä. Kuvassa 11 kuvataan pelihahmon päätöspuuta, jossa jokaisessa haarassa tehdään päätös hahmon yksilöllisiin tietoihin perustuen. Jokaisen oksan päässä on toiminto, jota siirrytään suorittamaan välittömästi, kun päätöspuu on käyty läpi ”juuresta oksaan”. (Millington – Funge 2009, 295-296.)



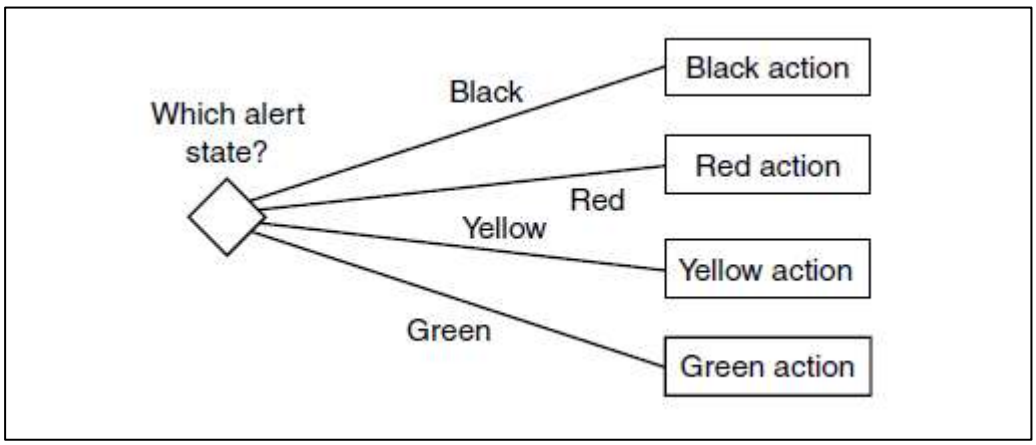
KUVA 11. Päätöspuu 1 (Millington – Funge 2009)

Jos hahmo näkee vihollisen, mutta etäisyyttä on yli 10 metriä eikä vihollinen sijaitse selustassa tai edessä, päätöspuun suorittama toiminto on ”Move” kuvan 12 mukaisesti.



KUVA 12. Päätöspuu 2 (Millington – Funge 2009)

Päätöspuita, joissa jokainen haara sisältää kaksi vaihtoehtoa, kutsutaan binäärisiksi päätöspuiksi, mutta vaihtoehtojen määrä on rajaton eli päätös voi johtaa yhteen useammasta toiminnosta kuvan 13 tavoin.

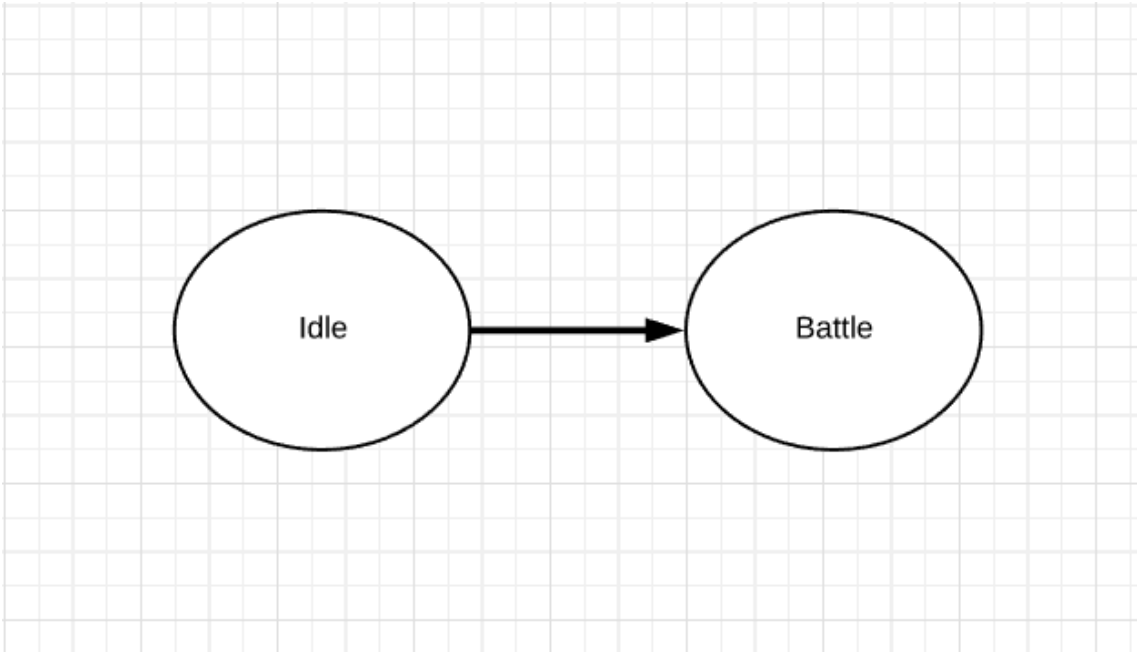


KUVA 13. Ei-binäärinen päätöspuu (Millington – Funge 2009)

State Machines eli tilakoneita käytetään usein terminä sulautetuissa järjestelmissä kuvaamaan digitaalisen järjestelmän tiloja ja niiden muutoksia (Digitaalipiirit/Tilakoneet.

2008). Peliohjelmoinnissa tätä voidaan soveltaa päätöksenteossa päätöspuiden yhteydessä. Tekoäly on aina jossain tilassa, joka määrää sen toiminnoista, niin pitkään kunnes tila vaihtuu. Tilakoneet voivat sisältää lisää tilakoneita niiden rakenteen mukaan. Esimerkiksi liikkuminen voi sisältää kiihdyttämistä, hidastamista tai kääntämistä. (Millington – Funge 2009, 309-310.)

Puretaan seuraavaan tilakone-esimerkkiin hirviötekoäly Monster Hunter -pelisarjasta (Capcom, 2004). Seuraava esimerkki on luotu pelkän tulkinnan pohjalta, koska avointa lähdekoodia pelille ei ole olemassa. Pelissä taistellaan tekoälyn ohjaamia hirviöitä vastaan. Pelaaja ja hirviö aloittavat metsästys tehtävän alussa eri paikoista. Hirviön tilakone aloittaa idle-tilasta eli hirviö vaeltelee pelialueella. Pelaajan astuminen hirviön näkökenttään laukaisee tilavaihdoksen, jolloin hirviö siirtyy idle-tilasta taistelutilaan kuvan 14 mukaisesti.



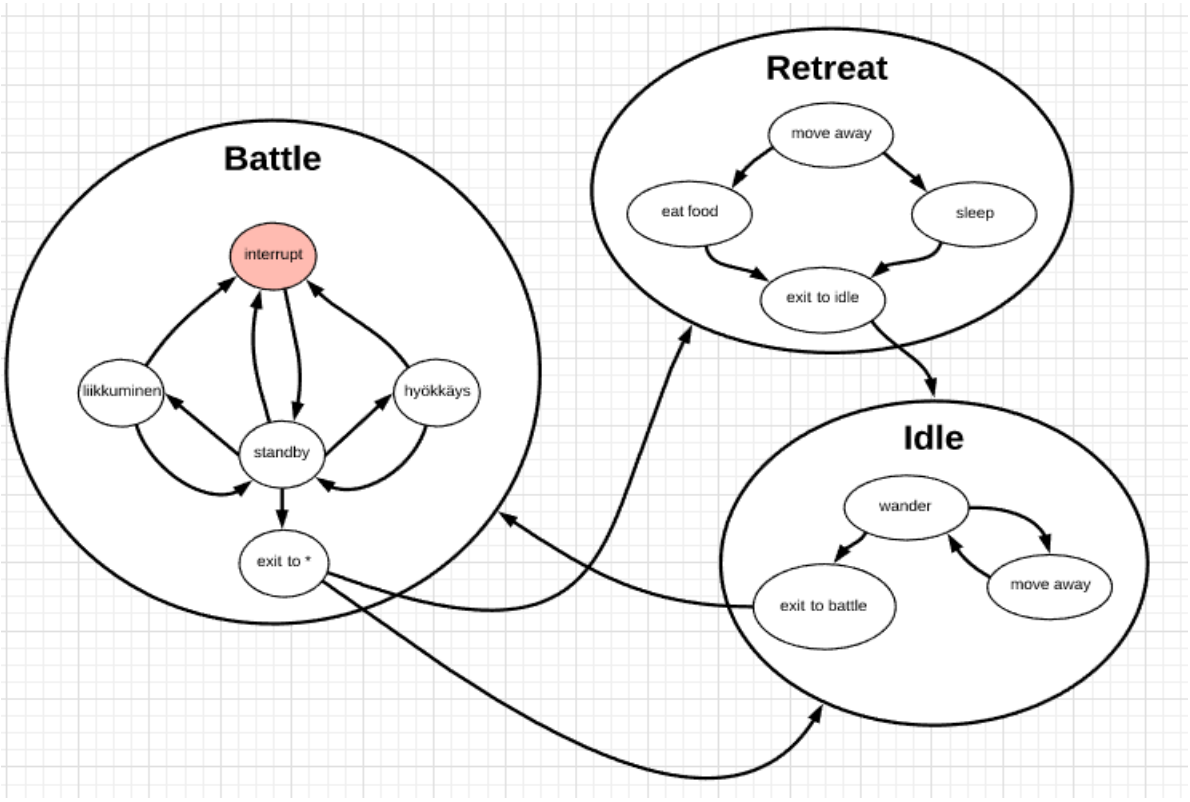
KUVA 14. Tilakoneen tilamuutos

Taistelutilassa hirviöllä on sisäinen tilakone, johon kuuluvat liikkuminen ja hyökkäykset. Kaikki tilat ovat yhteydessä kahteen yhteiseen tilaan. Kutsutaan näitä tiloja tässä erimerkissä termeillä ”standby” ja ”interrupt”. Tilat on esitetty kuvassa 15.

Standby-tila on vaihe, jossa tekoäly tekee tilanteeseen sopivan päätöksen. Ensisijainen tavoite hirviöllä on aiheuttaa pelaajalle vahinkoa. Jos pelaaja menettää kaikki elämä pis-

teensä, hirviö palaa idle-tilaan. Pelaajan ollessa hyökkäysten kantamien ulkopuolella hirviö liikkuu ensin lähemmäs ja palaa standby-tilaan tarkistamaan ollaanko vielä hyökkäysetäisyydellä. Jos hirviö on tarpeeksi lähellä pelaajaa, hirviön tila siirtyy hyökkäystilaan ja lopulta takaisin standby-tilaan.

Interrupt-tila on vaihe, johon hirviö siirtyy, kun pelaaja tekee tarpeeksi paljon vahinkoa tarpeeksi nopeasti. Interrupt-tilassa hirviön toiminnot keskeytyvät ja käynnistyy kärsimysanimaatio, jolloin pelaaja saa itselleen aikaa. Kuvan 15 tilakonekaavioon on lisätty retreat-tila, johon hirviö siirtyy, kun on kärsinyt tarpeeksi vahinkoa tai on väsynyt.



KUVA 15. Yksinkertainen tilakaavio hirviön taistelu-tilasta

Tietotekniikassa käytetään usein Boolen algebraa eli logiikkaa, jossa on arvo voi olla tosi tai epätosi eli 1 tai 0 (Beal 2018). **Fuzzy Logic** eli sumea logiikka on Boolen algebran vastakohta, jossa toden ja epätoden sijaan on useita eri arvoja toden ja epätoden välillä. Kaikkia arvoja toden ja epätoden välillä – eli 1 ja 0 – voidaan käyttää sumean logiikan avulla. (Prinkle 2016.) Ääriarvot voivat rajautua mille arvoille tahansa, esimerkiksi 0 ja

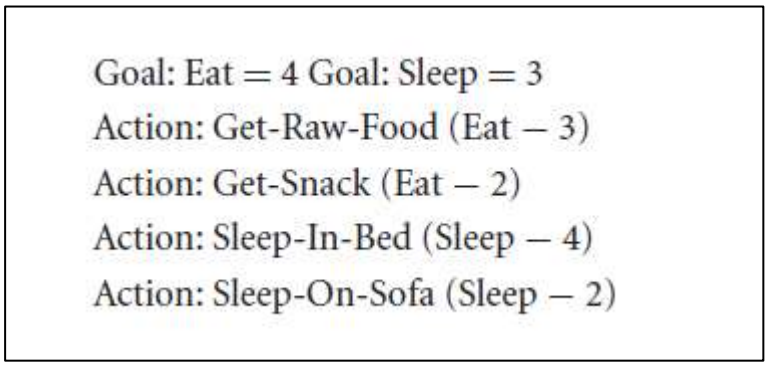
255, jolloin kaikki arvot – muuttujatyypistä riippuen – nollan ja 255 välillä ovat käytettävissä. Ääriarvojen sijoituksessa ei ole vaikuttavia tekijöitä logiikan ulkopuolelle. (Millington – Funge 2009, 372.)

Sumean logiikan toiminnallisuuteen liittyy erilaiset arvot, jotka edustavat hahmon tai asian tilaa tai muotoa. Hahmolla voi olla esimerkiksi arvot nälkä ja kunto. On tärkeää, että kaikkien näiden arvojen yhteenlaskettu summa vastaa maksimiarvoa. Oletetaan että ääriarvot ovat 0 ja 1, jolloin nälkä saa voi saada arvon 0,5 ja kunto 0,5. Nälkä voi saada myös arvon 0,3 ja tällöin kunto saa arvon 0,7. Näitä arvoja yhdessä kutsutaan sumeiksi ryhmiksi, englanniksi fuzzy sets. Hahmoilla voi olla useita eri ryhmiä vastaamassa erilaisista tiloista. (Millington – Funge 2009, 372.)

Sumea logiikka on käytännössä ohjain, joka auttaa päätöksenteossa. Sen rinnalla voidaan käyttää sääntöpohjaista ohjainta, joka sisältää tilanteeseen sopivat toiminnot. Esimerkiksi peliobjekti, joka seuraa toista liikkuvaa peliobjektia, voi sisältää arvot etäisyys ja jarrutus. Jarrutus on käänteisesti verrannollinen etäisyyteen eli jos etäisyys on ajan *t1* hetkellä 0,9 ja jarrutus on 0,1 niin ajan *t2* hetkellä etäisyyden ollessa 0,4 jarrutus on 0,6. Näille arvoille voidaan asettaa erikseen säännöt, joiden mukaan eri toiminnot toteutuvat. (Fuzzy Logic – Computerphile. 2014.)

Goal-oriented Behaviour eli tavoitepohjainen käyttäytyminen on laajakäsitteinen termi, joka käsittelee tekniikoita, joihin sisältyy tarpeiden ja tavoitteiden täyttäminen. Tavoitepohjaisen käyttäytymisen toteutukseen ei ole vain yhtä mahdollista tekniikkaa, vaan se voidaan toteuttaa usealla eri tavalla. (Millington – Funge 2009, 402.)

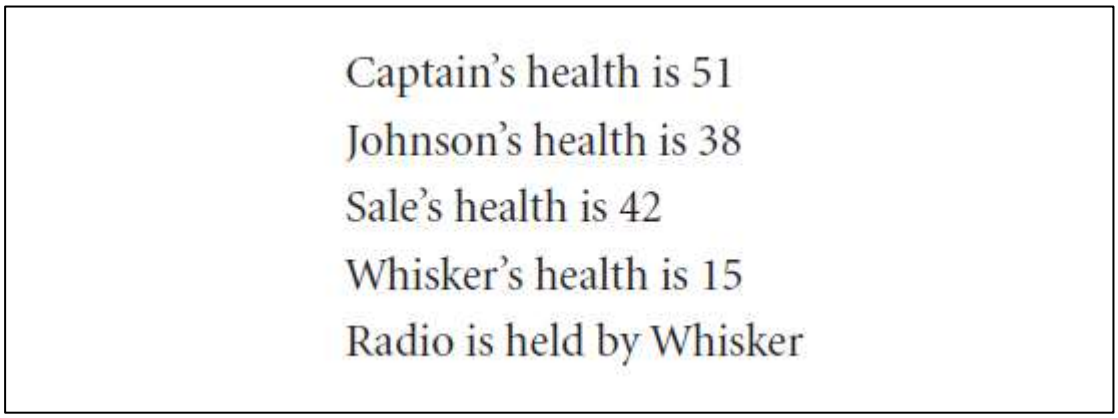
Hahmolla voi olla yksi tai useampi tavoite sekä useampi toiminto liittyen tavoitteisiin. Kuvassa 16 esitetään esimerkki, jossa kuvaillaan kaksi tavoitetta, Eat = 4 ja Sleep = 3, sekä neljä eri toimintoa.



KUVA 16. Goal-oriented Behaviour (Millington – Funge 2009)

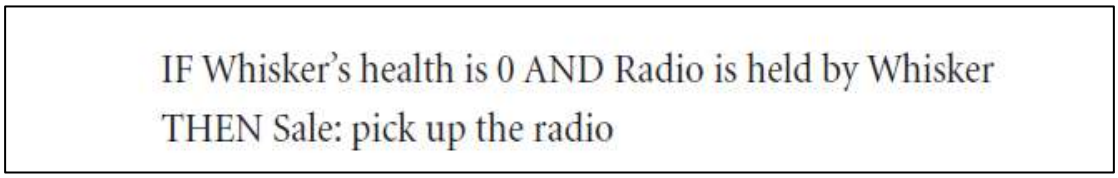
Helpoin keino saavuttaa molemmat tavoitteet olisi suorittaa toiminnot Get-Raw-Food ja Sleep-In-Bed koska niiden lupaamat instanssit joko täyttävät eniten tai kokonaan tavoite arvon. Näillä on siis suurin käytännön arvo. Tämä parhaimman heuristisen eli tilanteeseen sopivimman vaihtoehdon valinta voidaan toteuttaa käytännössä seuraavasti. Valitulle tavoitteelle valitaan paras toiminto siten, että kaikki vaihtoehdot vertaillaan keskenään, ja niistä valitaan suurimman käytännön arvon omistava toiminto. Tämä edellyttää, että toiminnot luodaan tietueina, jotka sisältävät niille annetun kiinteän käytännön arvon. Tämä on vain yksi tapa toteuttaa tavoitepohjainen käyttäytyminen. (Millington – Funge 2009, 403-404.)

Rule-Based System eli sääntöpohjainen ohjainrakenne on pitkäaikainen menetelmä videopelitekoälyissä. Olennaista tälle rakenteelle on, että se koostuu kahdesta osasta: hahmoa tai objektia koskevasta datasta sekä if-then-säännöistä. Säännöt seuraavat hahmoa tai objektia koskevaa dataa. Kun data täyttää tietyt if-ehdot, siirrytään suorittamaan then-toimintoja. If-säännöt sisältävät usein ehtoja, jotka pyrkivät vastaamaan tiettyjä pelihahmon data-arvoja. (Millington – Funge 2009, 428-429.) Kuvassa 17 on esitetty neljän pelihahmon elämäpisteet int-tyyppisinä muuttujina ja radion omistajaa osoittava muuttuja string-tyyppisen muuttujana.



KUVA 17. Miehistön elämäpisteet (Millington – Funge 2009)

Luodaan esimerkki narratiivi, jossa Whisker on kommunikointiasiantuntija ja hänellä on hallussaan radio. Asetetaan if-then-sääntö radion kantamisen suhteen eli radiota ja hahmojen elämäpisteitä koskeva sääntö. Kuva 18 esittää säännön: jos Whiskerin elämäpisteet ovat yhtä kuin nolla ja Whiskerillä on radio, niin Sale poimii radion itselleen. (Millington – Funge 2009, 429.)



KUVA 18. If-then-rakenne (Millington – Funge 2009)

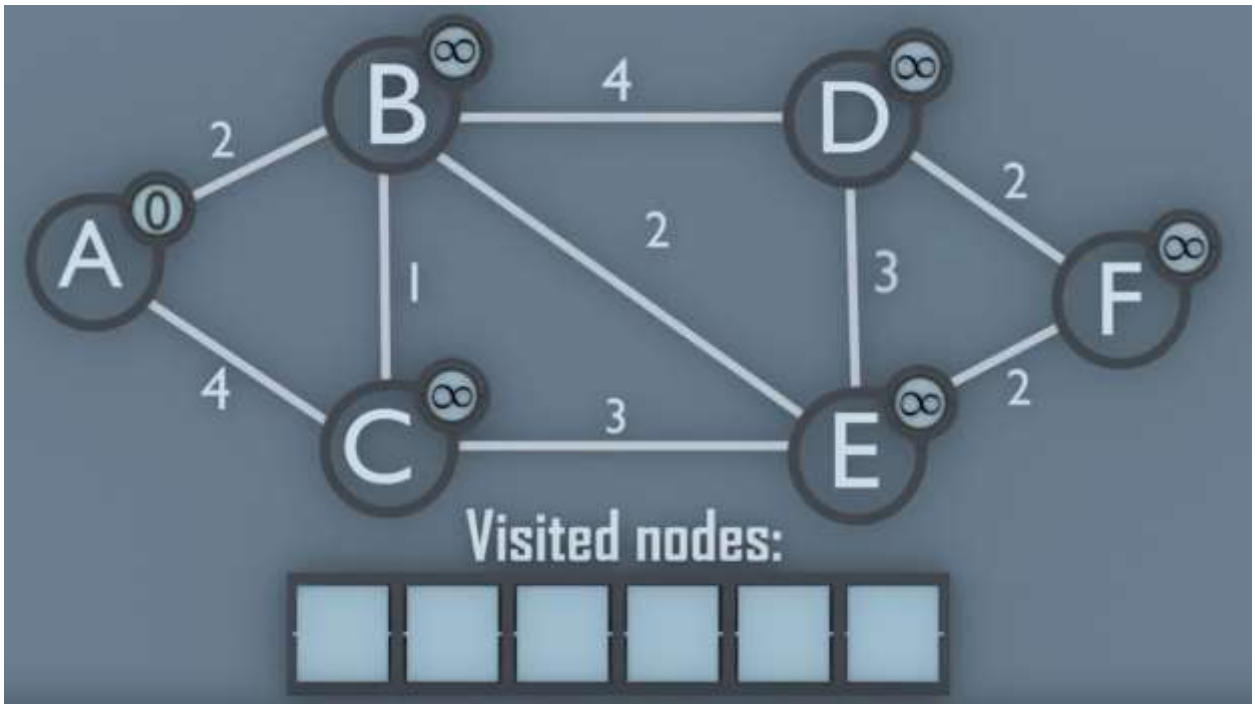
Jos hahmoja koskevat arvot täsmäävät sääntöjen vaatimuksien suhteen, sääntöpohjainen ohjainrakenne tekee ratkaisun, että Salen on poimittava radio. Täytyy huomioida, että tämä ei välttämättä voi toteutua, jos Salen data ei täsmää tiettyjen sääntöjen kanssa. Narratiivin mukaan Whisker on voinut esimerkiksi pudota jyrkänteeltä, jolloin Salella ei ole mahdollisuuksia poimia radiota, koska hän ei yksinkertaisesti voi päästä radion luokse. Tämän takia on tärkeää, että data, jota tämä ohjainrakenne käyttää, koskettaa vain hahmoa tai objektia, jota käsitellään, esimerkiksi faktoja, mitä pelihahmo tietää. On myös tärkeää, että dataa päivitetään näiden ohjainrakenteiden kanssa. Tilanne, jossa Sale poimii radion, muuttaa radion kantajan Whiskerista Saleksi. (Millington – Funge 2009, 430.)

2.5 Pathfinding-algoritmit

Pathfinding eli polunetsintää käytetään reittien etsimiseen sekä pelihahmojen ohjaamiseen peliympäristössä. On tilanteita, jolloin hahmon täytyy laskea vaivattomin reitti tason läpi pelaajan määrittelemään pisteeseen, valiten paras ja lyhin reitti. Parhaalla reitillä tarkoitetaan esteettömintä reittiä. Pathfinding ei sisällä erikseen päätöksen tekoon liittyviä algoritmeja, vaan keskittyy lyhimmän reitin laskemiseen. (Millington – Funge 2009, 197.)

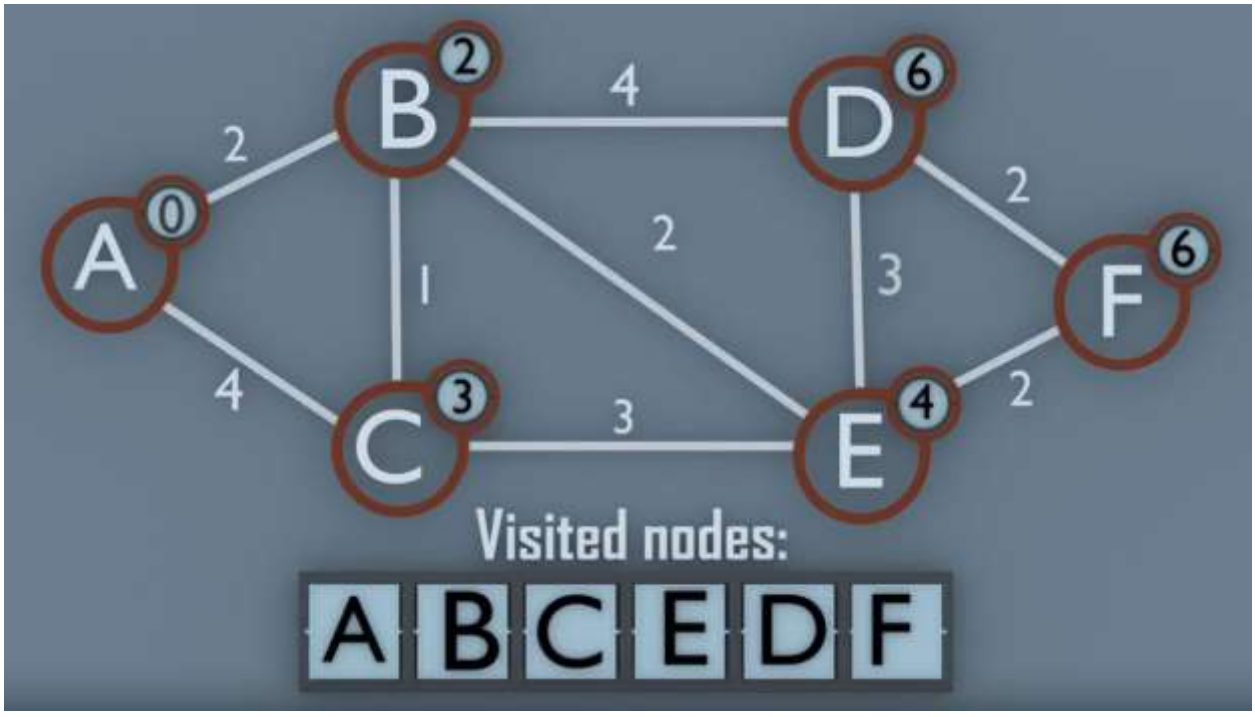
Yleisimpiä pathfinding-algoritmeja ovat **A*** sekä **Djikstra**. Molemmat suorittavat saman tehtävän, mutta toimivat toisistaan poikkeavasti. A* (luetaan A Star) on Dijkstran pohjalta luotu variantti, joka keskittyy ”parhaan ensimmäisen” tuloksen löytämiseen heuristiikan avulla (Difference and advantages between Dijkstra and A star. 2012). Heuristiikalla tarkoitetaan tilanteeseen lupaavinta vaihtoehtoa. Jos jokin vaihtoehto tuntuu tilanteeseen lupaavalta, se valitaan siitä huolimatta, mitä vaikutuksia sillä voi olla tuleviin tilanteisiin. (Poole – Mackworth 2010.)

Dijsktra, joka nimettiin Edsger Dijkstran mukaan, kehitettiin alun perin matemaattisten teorioiden ratkaisemiseen. Dikstran – kuten myös A*:n – toiminta perustuu paikannuspisteiden eli nodejen välisten arvojen yhteen laskemiseen. Kuvassa 19 on kuvattu kuusi nodea: A, B, C, D, E ja F. Näistä jokainen on yhdistetty kahteen tai useampaan vieressä olevaan nodeen. Näiden kaikkien nodejen välimatkoilla naapureihin on eri painoarvot, jotka kuvastavat ikään kuin matkan pituutta. Dijkstra aloittaa tutkimalla ensin viereisimmät nodet, jotka ovat yhteydessä jo tutkittuihin nodeihin. Kuvassa 19 ensimmäinen tutkittava node on A, jonka painoarvo on 0 – kyseessä on aloitusnode – ja muiden nodejen painoarvot ovat toistaiseksi äärettömiä. Kun node A on tutkittu eli sille on annettu arvo eli tässä tapauksessa 0, siirrytään tutkimaan nodet B ja C. A:n ja B:n välinen painoarvo on 2, joka lisätään A:n painoarvoon, joten B:n painoarvoksi saadaan 2. Node C saa painoarvokseen 4. Nodeista B ja C tulee nyt tutkittuja nodeja ja niiden viereiset tutkimattomat nodet käydään läpi seuraavaksi. (Jaakko 2016.)



KUVA 19. Node-kartta (Jaakko 2016)

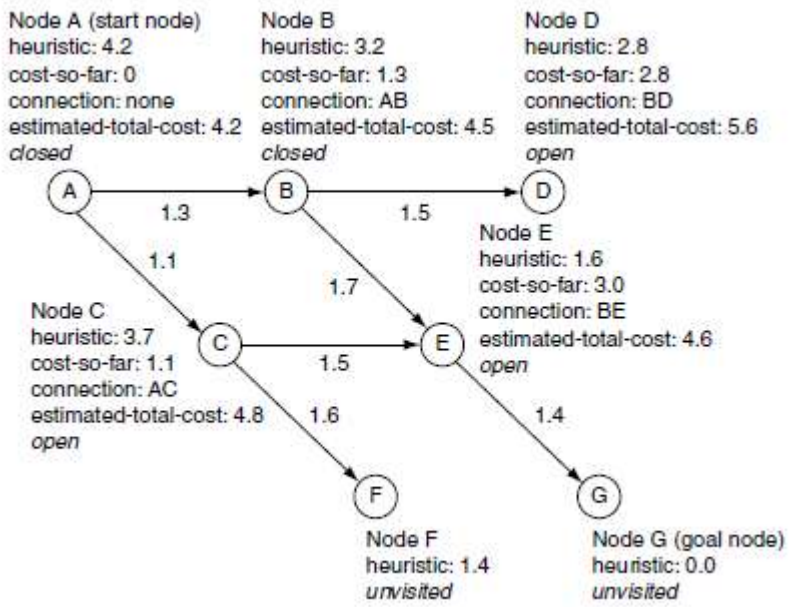
Kun kaikki nodet on tutkittu, lasketaan pienin yhdistetty painoarvo nodejen A ja F välillä. Tässä tapauksessa algoritmi kulkee kuvan 20 mukaan A–B–E–F.



KUVA 20. Nodejen arvot (Jaakko 2016)

Dijsktra käy läpi jokaisen mahdollisen noden, kunnes löytää reitin, toisin kuin A*, joka välttää heuristisen haun avulla nodet, jotka eivät ole oikeassa suunnassa tai ovat kelvottomia. Dijkstra tutkii myös nodet, jotka ovat päinvastaisessa suunnassa määränpäätä, ja tämä onkin merkittävin toiminnallinen ero verrattuna A*:iin. (Millington – Funge 2009, 204-215.)

Tarkastellaan, miten A* toimii heuristiikan avulla kuvan 21 avulla. Lähtöpisteenä toimii node A. A*-algoritmi tutkii ensin node A:n ympäröivät nodet. Täsmennetään vielä, että kaikilla nodeilla on välimatkan painoarvon lisäksi nyt heuristinen arvo, joka kuvastaa etäisyyttä tietystä nodesta määränpäähän. Tämä arvo ei muodostu yhteyksien painoarvojen mukaan, vaan voidaan kuvitella, että nodet edustavat vierekkäisiä kaupunkeja, mutta niiden välillä ei ole kulkuyhteyttä. Algoritmi tutkii seuraavana nodet B ja C, aivan kuten Dijkstra. B:n arvo kokonaisarvo ”estimated-total-cost” koostuu matkasta 1,3 ja heuristisesta arvosta 3,2, josta saadaan lopputulokseksi 4,5. C saa arvon 4,8. B on siis näistä nodeista suotavampi, joten algoritmi siirtyy tutkimaan sitä ja siirtää noden A käsiteltyjen nodejen joukkoon. Käsiteltyjä nodeja ei enää tutkita, eikä niiden arvoja muuteta. Node B:ssä tutkitaan sen viereiset nodet D ja E, jotka saavat arvot 5,6 ja 4,6. (A* (A Star) Search Algorithm – Computerphile. 2017.) Tässä huomataan jo ero Dijsktraan: siinä missä Dijsktra tutkii kaikki nodet, A* tutkii vain heuristisesti lupaavimmat nodet, mikä ilmenee käytännössä nopeampana toimintana verrattuna Dijsktraan.



KUVA 21. A:n toiminta (Millington – Funge 2009)*

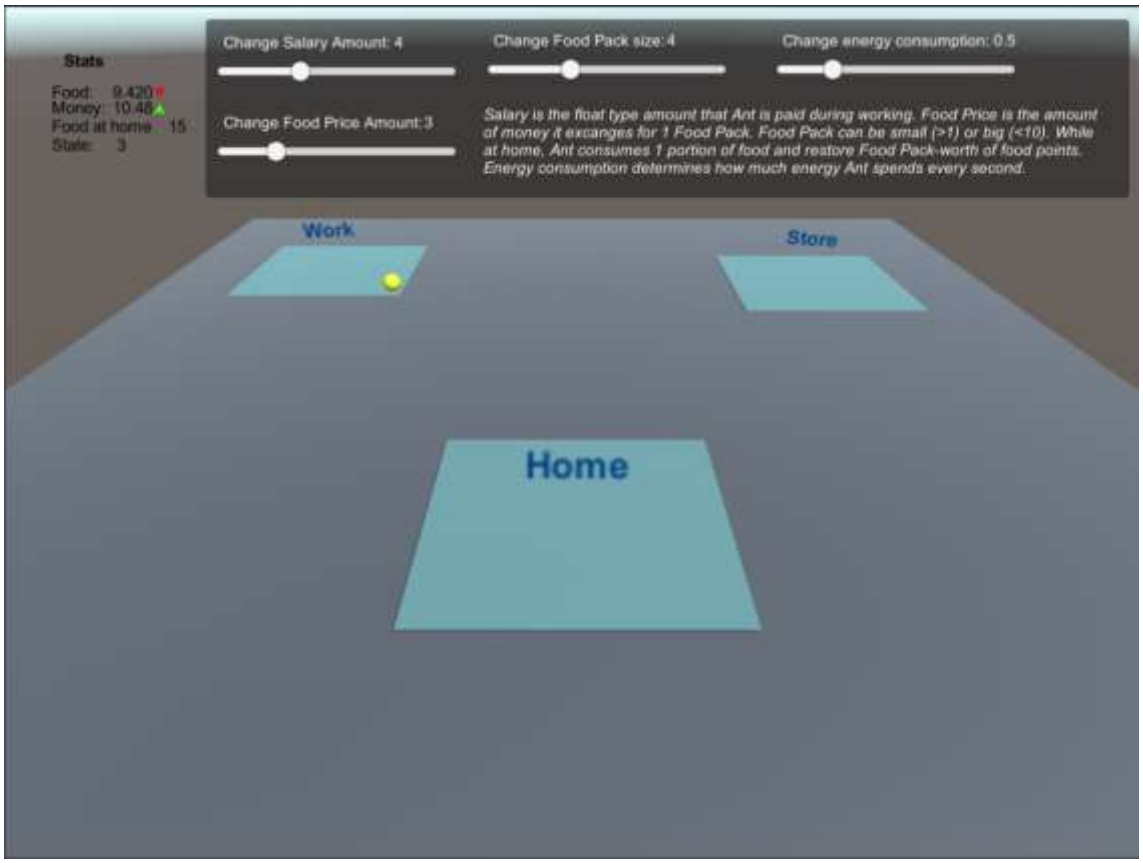
Dijkstran ja A*:in lisäksi on useita muita pathfinding-algoritmeja, joista useat ovat variant-teja edellä mainituista algoritmeista. (Millington – Funge 2009, 215.)

3 TERRARIUM-PROJEKTI

Osana tätä opinnäytetyötä toteutettiin tekninen toteutus, joka kantoi työnimeä Terrarium-projekti. Tekninen toteutus sisältää kaksi erilaista tekoälyä, joita voidaan soveltaa esimerkiksi NPC-hahmojen ohjelmoinnissa. Ensimmäinen toteutus on päätöksien tekoon keskittyvä Ant ja toinen A*-algoritmiä käyttävä Adventurer. Toteutus on kirjoitettu C#-ohjelmointikielellä ja toteutettu Unity-pelimoottorilla.

3.1 Ant-toteutus

Ensimmäinen kokonaisuus käsittelee tilakoneen toimintaa neljän eri tilan välillä: idle, seek home, seek store ja seek work. Ohjelmassa on Ant-olio, jolla on kolme tärkeää float-tyyppistä muuttujaa: Food (ruoka), Money (raha) ja Food at home (ruokavarasto). Tilakone pohjautuu näiden tarpeiden täyttämiseen siten, että Ant liikkuu 3D-maailmassa joko kotiin, töihin tai kauppaan. Kotona Ant voi syödä ruokaa varastosta ja nostaa omaa food-tasoaan. Töissä Ant kerää rahaa, jonka se voi kuluttaa kaupassa ruokavaraston täyttämiseen. Kokonaisuudessa painottuu myös päätösten teko tiettyjen ehtojen täyttyessä. Kuvassa 22 esitetään Ant toiminnassa.



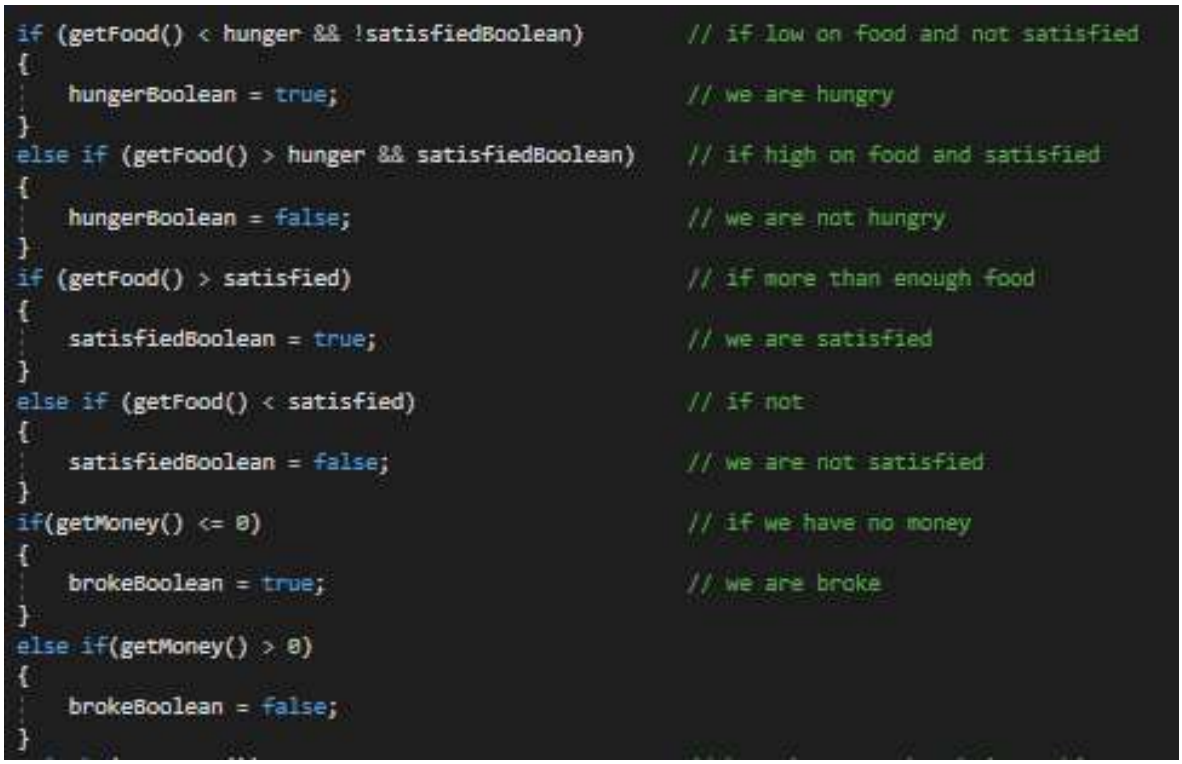
KUVA 22. Ant projektissa Terrarium

Kun Ant on olemassa, sen Food-arvo putoaa joka sekunnin välein 0,5:llä. Tämä toteutettiin yksinkertaisesti Unityn DeltaTime-funktiolla. Statistiikka-arvojen muuttaminen tapahtuu aluekohtaisesti. Kotona Food-arvo nousee kaksi yksikköä sekunnissa, mikä ilmenee ohjelmassa päinvastaisena 1,5-kertaisena Food-arvon kasvuna. Tämä myös vähentää ruokavarantoja kahdella yksiköllä sekunnissa. Kaupassa Food at home eli ruokavarannot nousevat kolmella sekunnissa, mutta vähentävät Antin Money-arvoa ruoan hinnalla – tässä tapauksessa neljällä joka sekunti. Yleisin ongelma Antin kanssa oli tasapainottaa ympäristömuuttujat, mikä olisi pidentänyt Antin elinaikaa. Usein Ant päätyi tilanteisiin, joissa sen täytyi valita kahden tarpeen välillä, mutta tarpeet kääntyivät päittäin, jolloin Ant jäi värisemään kahden alueen väliin pyrkien kumpaankin vuorotellen. Antin ennenaikainen kuoleminen saatiin korjattua lisäämällä UI-ohjain, joka sisältää liukusäätimiä säätelemään ympäristön muuttujia. Näitä muuttujia ovat Salary (palkka), Food price (ruoan hinta) ja Food Pack size (ruokapaketin koko). Lisäksi yksi liukusäädin on varattu Antin energian kulutuksen säätelyyn. Ympäristön muuttujia voidaan säädellä kesken ohjelman eli ne muuttuvat dynaamisesti.

Antin päätöksen teko toteutettiin boolean-muuttujilla sekä switch-case-valintarakenteella, joka parametrin mukaan määrittää ohjelmavaiheen, mitä käytetään. Switch-case-rakennetta käytettiin luomaan tilakone, joista jokainen tila käskee Antia liikkumaan eri paikkaan. Tila 1 ohjaa Antin kotiin, tila 2 kauppaan ja tila 3 töihin käyttäen MonoBehavior-luokan Vector.Lerp-funktiota. Täytettävät kriteerit ja ehdot tilavaihdokselle vaihtelevat tilojen kesken. Kuvassa 23 kuvataan, millaisilla ehdoilla Ant vaihtaa näitä tiloja. Tila 0 edustaa Antin idle-tilaa, joka sisältää tilavaihdon mahdollisuuden kolmeen muuhun tilaan. Idle-tila muuttuu tilaan 1, jos hungerBoolean on tosi, tilaan 2, jos kotona ruoan määrä on 0 tai vähemmän, ja tilaan 3, jos satisfactionBoolean tai brokeBoolean ovat jompikumpi totta. HungerBoolean, satisfactionBoolean ja brokeBoolean määräytyvät kuvan 24 mukaisesti.

```
switch (getState())                                 // here happens the choice making
{
    case 0:
        //idle
        if (hungerBoolean)                          // if hungry, go home
        {
            changeState(1);
        }
        if (homeScript.getFoodSupplies() <= 0)      // if there's no food, go buy it
        {
            changeState(2);
        }
        if (satisfiedBoolean || brokeBoolean)       // if satisfied OR broken, go to work
        {
            changeState(3);
        }
        break;

    case 1:
        //seek home
        moveTo("home");
        if (homeScript.getFoodSupplies() <= 0)      // if there's no food, go buy it
        {
            changeState(2);
        }
        else if (satisfiedBoolean || brokeBoolean)  // if satisfied OR broke, go to work
        {
            changeState(3);
        }
        break;

    case 2:
        //seek store
        moveTo("store");
        if (hungerBoolean && homeScript.getFoodSupplies() >= 5) // if hungry AND food at home, go home
        {
            changeState(1);
        }
        else if (brokeBoolean)
        {
            changeState(3);                         // if broke, go to work
        }
        break;

    case 3:
        //seek work
        moveTo("work");
        if (hungerBoolean && !brokeBoolean)         // if hunger BUT NOT broken, go to home
        {
            changeState(1);
        }
        else if (homeScript.getFoodSupplies() <= 5 && !brokeBoolean && getMoney() >= storeScript.getFoodPrice())
        {
            changeState(2);                         // if there's low food at home AND Ant has money
        }                                           //  AND it has more than enough money
        break;                                      // to buy food, go to store
```

KUVA 23. Antin tilakoneen toteutus switch-case-valintarakenteella



```
if (getFood() < hunger && !satisfiedBoolean)        // if low on food and not satisfied
{
    hungerBoolean = true;                           // we are hungry
}
else if (getFood() > hunger && satisfiedBoolean)    // if high on food and satisfied
{
    hungerBoolean = false;                          // we are not hungry
}
if (getFood() > satisfied)                          // if more than enough food
{
    satisfiedBoolean = true;                        // we are satisfied
}
else if (getFood() < satisfied)                     // if not
{
    satisfiedBoolean = false;                       // we are not satisfied
}
if(getMoney() <= 0)                                 // if we have no money
{
    brokeBoolean = true;                            // we are broke
}
else if(getMoney() > 0)
{
    brokeBoolean = false;
}
```

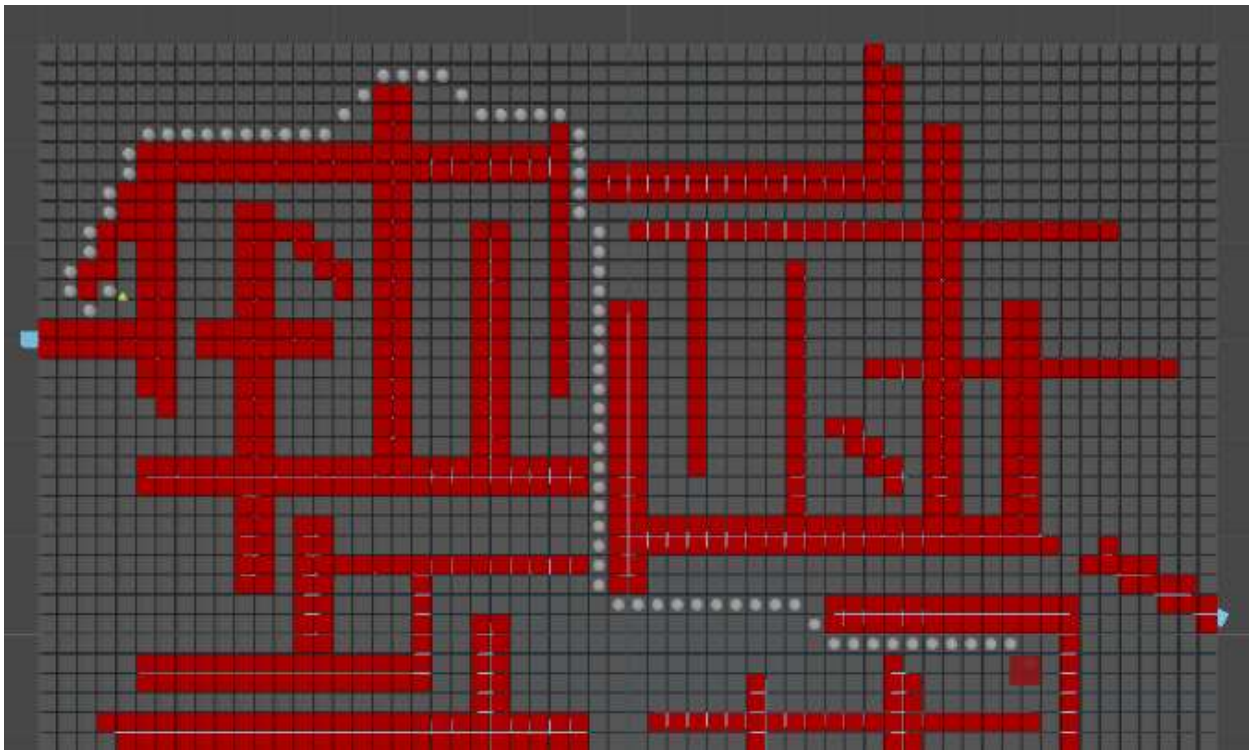
KUVA 24. Antin booleanien ehdot

Käyttötarkoitus Antin päätöksentekotekoälylle voisi olla esimerkiksi The Sims -tyylisen pelin NPC-hahmoille, joilla on selkeät tarpeet. Nämä tarpeet vaihtelisivat ympäristön ja pelaajan tekemien päätösten mukaan.

3.2 Adventurer-toteutus

Teknisen toteutuksen toinen osa käsittelee polunetsintää kaksiulotteisessa ympäristössä. Työstövaiheessa käytin apunani Sebastian Laguen Youtube-videosarjaa A* Pathfinding (Lague 2014). Toteutus koostuu kaksiulotteisesta peliympäristöstä ja kolmesta tärkeästä peliobjektista, jotka on nimetty Adventurer, Treasure ja A*. Kokonaisuutta ohjaa A*-peliobjekti, joka sisältää ruudukon alustuksen sekä polunetsintäalgoritmin toiminnallisuuden, jonka tuloksena on polku Adventurer- ja Treasure-objektin välillä. A* sisältää Grid-skriptin, joka ottaa halutun ruudukon koon int-tyyppisinä parametreina, noden koon float-tyyppisenä parametrina ja lopuksi alustaa ruudukon A*-objektin ympärille. Ruudukon alustuksessa huomioidaan A*-objektia ympäröivät peliobjektit, jotka ovat erikseen asetettu kulkukelvottomiksi. Kuvassa 25 on prototyyppitoteutus, jossa ruudukko koostuu tum-

manharmaista kuljettavista ja punaisista kulkukelvottomista nodeista. Adventurer on sijoitettu oikeaan ylänurkkaan ja Treasure vasempaan alanurkkaan ja näiden välille jokaiselle A*:in valitsemalle nodelle on annettu vaaleanharmaa välietappi piste. Vaaleanharmaan pisteet muodostavat lyhimmän reitin Adventurerin ja Treasuren välille ja edustavat A*:in laskema reittiä.

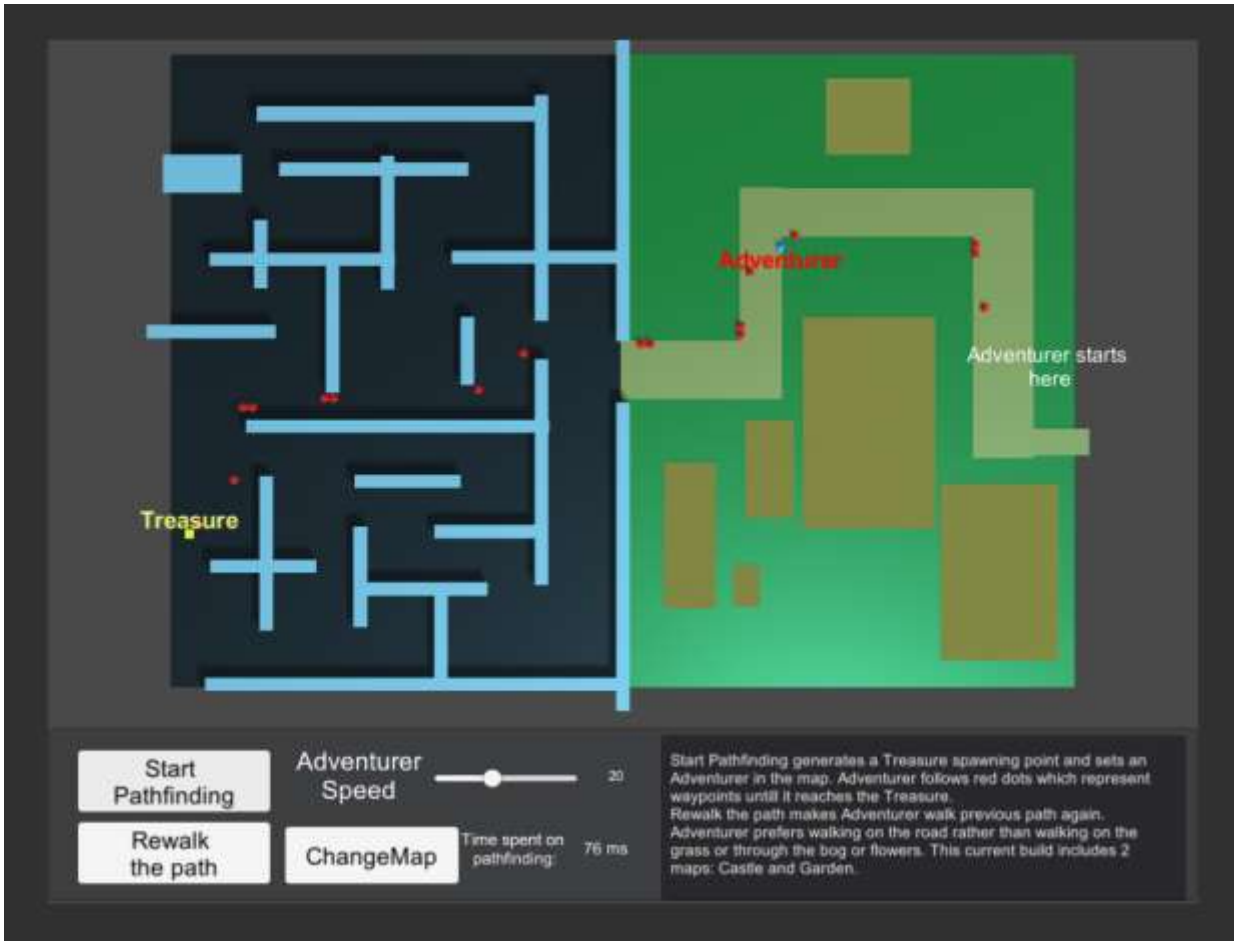


KUVA 25. Reitti Adventurerin ja Treasuren välillä

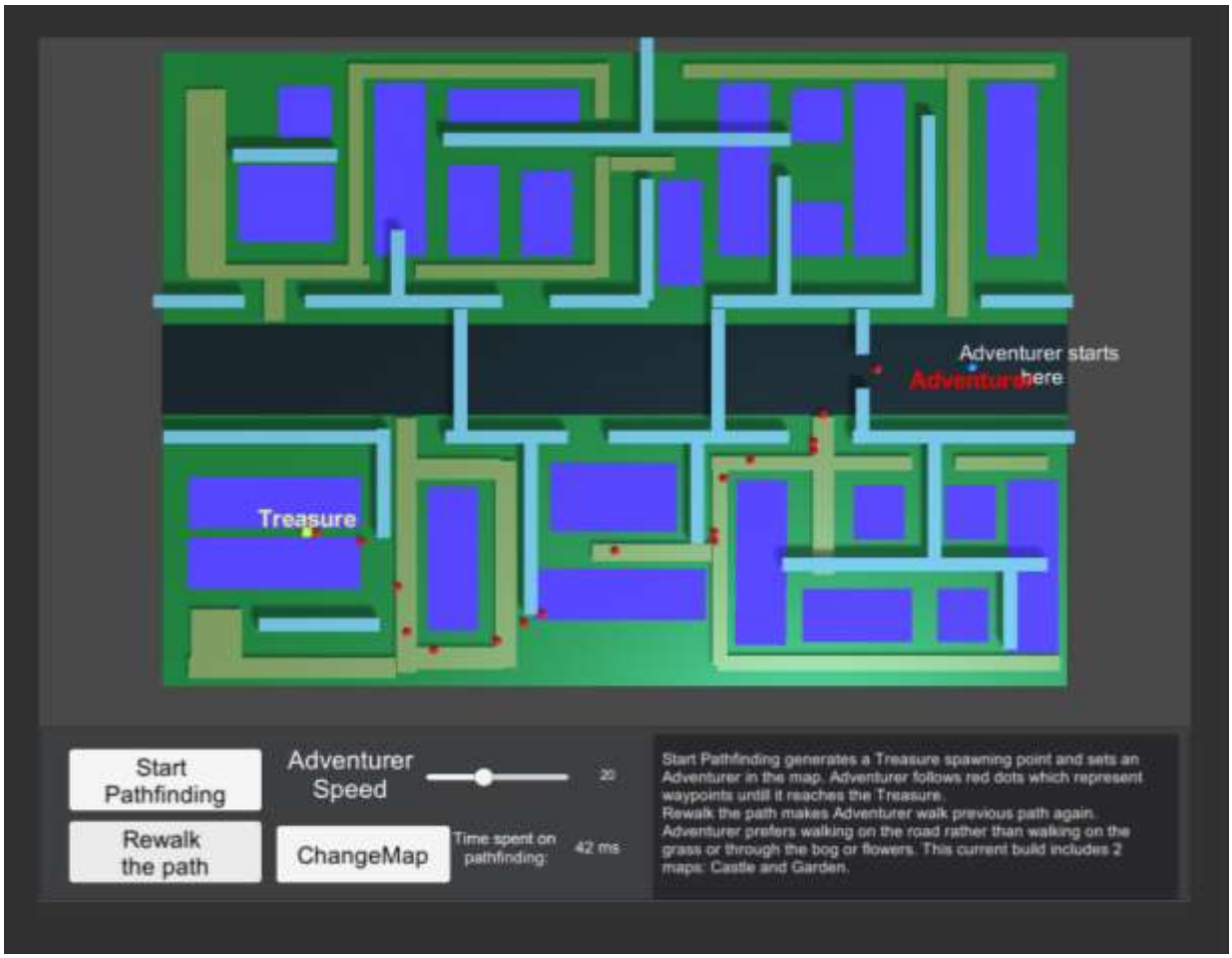
Ruudukon jokainen node on luotu Node-luokasta ja sisältää x- ja y-koordinaatit, boolean-tyyppisen walkable-muuttujan sekä int-tyyppisen penality muuttujan. Walkable muuttujaa käsitellään ruudukossa jokaista ympäristöobjektia kohden. Kuvan 25 punaiset nodet vastaavat seiniä, jotka on asetettu Unityn Layer -hierarkiaan tasolle ”Unwalkable”, ja jokainen node, joka sisältää tälle tasolle kuuluvan peliobjektin, on kulkukelvoton.

Valmiissa toteutuksessa node-listaa yksinkertaistetaan siten, että kaikki samansuuntaiset nodet seulotaan pois. Luodaan uusi lista, johon lisätään node, vain jos kyseessä on käännös. Tämä uusi lista välitetään sitten Adventurerille, jolloin Adventurer ei pysähdy jokaisen suoralla linjalla sijaitsevan noden kohdalla, vaan liikkuu suoraan niin pitkään, kunnes tapahtuu käännös.

Adventurer-toteutus sisältää UI-ohjaimen, jolla voidaan aloittaa polunetsintä, kävellä polku uudestaan, vaihtaa kenttää ja säädellä Adventurerin nopeutta. Lisäksi UI-ohjain kertoo, kuinka kauan polunetsintään käytettiin aikaa millisekunneissa. Kuvassa 26 on kenttä Castle ja kuvassa 27 kenttä Garden. Siniset suorakulmiot edustavat seiniä, tumman siniset laatoitettua lattiaa, kirkkaan vihreät nurmikkoa, harmaan vihreät tietä ja ruskeat sekä violetit pensaita. Jokaisella alueella on oma penality-muuttujansa, joka vaikuttaa Adventurerin suosimaan reittiin.



KUVA 26. Adventurer-toteutuksen Castle-kenttä



KUVA 27. Adventurer-toteutuksen Garden-kenttä

4 YHTEENVETO

Opinnäytetyön tarkoituksena oli tutkia, millaisia eri tekoälyjä videopeleissä käytetään sekä käydä läpi teknisentoteutuksen yksityiskohtia.

Mielestäni työ saavutti tavoitteet, mutta sisältö kävi läpi muodonmuutoksen. Erilaisten tekoälyjen tutkiminen täsmentyi erilaisiin algoritmeihin, joita peleissä voidaan hyödyntää osana tekoälyjen toteutusta. Työssä käsiteltiin kahden eri pelin tekoälyjä ja pilkottiin niiden kokonaisuuksia pienempiin osiin, sen sijaan, että olisin esitellyt useampia erilaisten genrejen pelitekoälyjä. Teoriaosuudessa esiteltiin erilaisia menetelmiä ja niiden algoritmeja, jotka esiintyvät osittain teknisessä toteutuksessa.

Koin haastavaksi kuvailla tekoälyjen käyttöä videopeleissä, ja niiden esittely teoreettisesti oli todella hankalaa. Kaikki kokemukseni pelitekoälyihin koostuvat omista vuorovaikutuskokemuksistani videopelien parissa. Esimerkiksi ympäristön rehevöittämiseen pohjautuvaa teoriatekstiä en onnistunut löytämään, jolloin näin oman tulkintani tässä tapauksessa aiheelliseksi. Algoritmeihin tutustumisessa oli haastetta yrittää nimetä englanninkieliset termit suomeksi, mutta joidenkin termien kanssa en voinut muuta kuin tyytyä englanninkieliseen termiin.

Koen että työ onnistui hyvin ja opin sitä tehdessä paljon polunetsinnästä, päätöksenteosta ja erilaisista algoritmeista, mitä peliohjelmoinnissa voidaan hyödyntää.

LÄHTEET

Artificial Intelligence (AI). 2018. Technopedia. Saatavissa: https://www.techopedia.com/definition/190/artificial-intelligence-ai. Hakupäivä 6.2.2018.

Artificial Intelligence (video games). 2018. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Artificial_intelligence_(video_games).Hakupäivä 12.2.2018.

A* (A Star) Search Algorithm – Computerphile. 2017. Computerphile. Video. Saatavissa: https://www.youtube.com/watch?v=ySN5Wnu88nE. Hakupäivä: 28.2.2018.

Beal V. 2018. Boolean logic. Webopedia Saatavissa: https://www.webopedia.com/TERM/B/Boolean_logic.html. Hakupäivä 21.3.2018.

Difference and advantages between Dijkstra and A star. 2012. Stackoverflow. Keskustelupalsta. Saatavilla: https://stackoverflow.com/questions/13031462/difference-and-advantages-between-dijkstra-a-star. Hakupäivä 27.2.2018.

Digitaalipiirit/Tilakoneet. 2008. Wikikirjasto https://fi.wikibooks.org/wiki/Digitaalipiirit/Tilakoneet. hakupäivä 5.2.2018.

Fuzzy Logic – Computerphile. 2014. Computerphile. Video. Saatavissa: https://www.youtube.com/watch?v=r804UF8Ia4c. Hakupäivä: 22.3.2018.

Grand Theft Auto. 2018. Wikipedia. Saatavissa: https://fi.wikipedia.org/wiki/Grand_Theft_Auto. Hakupäivä 23.3.2018.

Immersio. 2017. Wikipedia. Saatavissa: https://fi.wikipedia.org/wiki/Immersio Hakupäivä 20.2.2018

Jaakko. 2016. Dijkstra's algorithm explained. Video. Saatavissa: https://www.youtube.com/watch?v=CL1byLngb5Q. Hakupäivä 27.2.2018.

Lague S. 2014. A* Pathfinding (E01: algorithm explanation). Video. Saatavissa: https://www.youtube.com/watch?v=-L-WgKM-FuhE&list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW. Hakupäivä 13.3.2018.

Millington I. – Funge J. 2009. Artificial Intelligence for games. San Francisco: Morgan Kaufmann.

Narratiivi. 2018. Wikipedia. Saatavissa: https://fi.wikipedia.org/wiki/Narratiivi. Hakupäivä 26.2.2018.

Polunetsintä. 2013. Wikipedia Saatavissa: https://fi.wikipedia.org/wiki/Polunetsint%C3%A4. Hakupäivä 12.2.2018.

Poole, D. – Mackworth, A. 2010. Heuristic search. Artificial Intelligence: Foundation of Computational Agents. Cambridge: Cambridge University Press. Saatavissa: http://artint.info/html/ArtInt_56.html. Hakupäivä 27.2.2018.

Prinkle M. 2016. How is Fuzzy Logic used in game development. Quora. Saatavissa: https://www.quora.com/How-is-Fuzzy-Logic-used-in-game-development. Hakupäivä 21.3.2018.

Reynolds Craig W. 1999. Steering Behaviors for Autonomous Characters. Proceedings of Game Developer Conference. Saatavilla: https://www.red3d.com/cwr/papers/1999/gdc99steer.pdf. Hakupäivä 20.2.2018.